Arkestra Documentation

Release dev

Daniele Procida

Sep 29, 2017

Contents

Arkestra is an intelligent, semantic web web publishing system for organisations and institutions.

Arkestra builds on Django CMS with modules that handle contacts & people, news & events, automatic image publishing and management, maps and more. At the same time it uses a completely standard version of Django CMS, and can easily be added to an existing Django CMS installation.

It is flexible, modular and extensible, and has a rich set of automated web publishing tools.

Arkestra was developed at Cardiff University School of Medicine to provide a platform for the School's web presence.

Arkestra can be obtained from our GitHub repository.

CHAPTER 1

Getting started

Installation

Quick start

Prerequisites

These steps assume that you have the following available on your system:

- pip
- Git
- Mercurial

Set up a development environment

```
git clone https://github.com/evildmp/Arkestra.git
cd Arkestra
make develop
source .env/bin/activate
```

From this point you can wrestle Arkestra into submission on your own, or get started using the *example* project included.

Set up the supplied example project

Arkestra includes an example project for Django 1.4.

Assuming that your code was installed into src/arkestra in your virtualenv:

```
cd example
python manage.py syncdb --noinput --all # set up a new database; don't prompt for␣
↪superuser and use syncdb even on applications with migrations
python manage.py loaddata example_database.json # load the example database from the␣
↪fixture
```

Fire up the server

```
python manage.py runserver 0.0.0.0:8000 # go!
```

You should see the famous Institute of Mediaeval Medicine website, complete with images and all kinds of interesting content.

Username and password are both *arkestra*.

Note!

When you start up the server, you may not see any of the news/events/vacancies/studentships items you'd expect.

That's because they're out of date by now - this database was created some time ago.

Go into the news/events/vacancies/studentships and give them more appropriate dates.

Run tests

For tests, pytest test framework with nice pytest-django plugin is used instead of `standard` unittest approach recommended by Django.

To run tests:

```
py.test tests
```

For multipython testing, use just:

```
tox
```

To see test coverage:

```
tox -e coveralls
```

More notes about installation

With luck you won't even need to refer to this, but just in case, here it is.

The Arkestra applications

Arkestra is a collection of applications, each of which needs to be put on your PYTHONPATH (Arkestra's setup.py should do this for you):

- arkestra_image_plugin
- arkestra_utilities

- contacts_and_people
- housekeeping
- links
- news_and_events
- vacancies_and_studentships
- video

Other components

Akestra requires installation of various components (Arkestra's setup.py should do this for you). They include:

- Django CMS
- Django Filer
- Django Widgetry
- Semantic Presentation Editor
- django-polymorphic
- BeautifulSoup
- django-typogrify
- pyquery
- easy-thumbnails
- django-appmedia
- PIL

pip freeze

This is what *pip freeze* reports, just for your information:

- -e git+git@github.com:evildmp/Arkestra.git@a4cba94661af4b22025a301e70c1c5438bbecd65#egg=Arkestra-dev
- BeautifulSoup==3.2.1
- Django==1.4
- Pillow==1.7.7
- South==0.7.5
- cssselect==0.7.1
- django-classy-tags==0.3.4.1
- -e git+git@github.com:divio/django-cms.git@56afb01890396fe0562c3ce6700afbff81487f80#egg=django_cms-dev
- -e git+https://github.com/stefanfoulis/django-filer.git@52aa62b11b7c890e45ed0eb45ef37d0300d5d3ee#egg=django_filer-dev
- django-mptt==0.5.2
- django-polymorphic==0.2

- django-sekizai==0.6.1
- django-typogrify==1.3
- -e git+git@github.com:evildmp/django-widgetry.git@87885698bb3b1c0913a642a0b242a21557b3da09#egg=django_widgetry-dev
- easy-thumbnails==1.0.3
- html5lib==0.95
- lxml==2.3.4
- pyquery==1.2.1
- -e hg+https://bitbucket.org/spookylukey/semanticeditor@6a344716ddd2791c98af773fdfc6bb50107c3b8f#egg=semanticeditor-dev
- smartypants==1.6.0.3
- wsgiref==0.1.2

Pillow and PIL

Arkestra will install Pillow. It'll work with PIL too, but Pillow is much easier to use with setuptools.

If you must use PIL

You can try *pip install PIL*, but it doesn't always seem to work very well.

This is often because pip installs PIL from source, and if you don't have the development packages (C headers) then it won't compile it with support for all required file formats. Make sure you get the *-dev* packages first. On a Debian system doing *apt-get install libjpg-dev libpng-dev* before getting PIL via pip fixed a big problem with image uploads because thumbnails weren't being generated.

Starting your own project

Using the bundled example project

Arkestra comes with a bundled example project - in the *example* folder - which is ready to go, complete with database and media files.

Once you have the example project running, have a look at the site.

The admin interface is at */admin*; username and password are both *arkestra*.

Once logged in to the admin, set your *Site* appropriately - */admin/sites/site/*.

Running your own project

The *example* project in Arkestra contains a ready-made *settings.py* file, not to mention the *urls.py* and so on that you'll need.

Either copy these, or if you know what you're doing, copy the relevant parts to the files in your own project.

Start with the Python runserver, and get that going.

In order to make anything work, you'll need to do log in to Admin (username and password are both *arkestra*) and do three things.

- create a Page
- create an Entity
- link the Entity to the Page, by selecting the Page as the Entity's /Home page/

And since Arkestra needs to know what the *base entity* of your site is, in your `arkestra_settings` file add something like:

```
ARKESTRA_BASE_ENTITY = 1
```

where the value corresponds to the id of your base entity.

In production

Run *collectstatic* - https://docs.djangoproject.com/en/dev/ref/contrib/staticfiles for media files to get them into the right place.

For deployment, point your web hosting platform not at *settings.py*, but *deployment_settings.py*, which turns off various debug modes.

CHAPTER 2

Arkestra for users

The concept of Arkestra

Arkestra was conceived as an answer to the problem: **what is the best way for an organisation to publish information on the web?**

The shortcomings of web content management systems

Content management systems rarely work very well, especially when organisations use them, and the bigger the organisation and the site, the worse the problems.

Web content management systems waste users' time, and their work, and produce sites full of inconsistent content, inconsistently presented.

They don't offer users enough freedom to do what they need, and they offer too much, so they can do things they shouldn't - both at the same time.

Arkestra is an attempt to avoid these pitfalls, by taking a different approach.

Arkestra's approach

Arkestra **models the real world**, building meaningful real-world relationships into its structures.

It's a **semantic publishing system** rather than a *content management system*; it handles **meaningful information**, rather than mere content or data, and makes as much use as possible of this information as possible.

Arkestra has been designed around a single key imperative: **don't waste people's time.**

Wherever possible, Arkestra should:

- make it easy to capture and manage useful information
- make the information easily re-usable
- re-use it automatically and appropriately whenever possible

Arkestra's model of the world

Arkestra begins by modelling the basic concept of an organisation. An organisation - an **Entity** in Arkestra's terminology - might:

- contain other sub-entities within it
- occupy a number of buildings across different sites
- have people who have various different roles in it, and engage in different activities
- hold events of various kinds
- need to publish news about its activities

... amongst numerous other things.

Arkestra models all of these, and their relationships to each other.

If an entity is a based in a certain building, and that building has a postal address, Arkestra can work out the correct postal address for the entity. If a person works in that entity, Arkestra can infer the person's address.

If a particular set of web pages is associated with a particular entity, then Arkestra will be able to associate other relevant things with those pages - people, events, news and so on - automatically.

These things can will also be associated with each other in various important ways in the real world: an event might feature particular people, and take place at a particular date and time, in a particular location; Arkestra models and makes use of these relationships too.

The next section will examine entities more closely.

Entities

What exactly is an entity?

An *entity* could be:

an organisation: *Cardiff University School of Medicine*

a part of a larger organisation: *Institute of Infection & Immunity Research Office*

a group of other entities: *Research institutes*

Administrative & support offices

a collection of people: *Web editors School IT liaison officers*

Entities exist in a *hierarchy*, which might look like:

- Cardiff University
  - Cardiff University School of Medicine
    - Institutes
      - Institute of Infection & Immunity
      - Institute of Cancer Genetics
      - Institute of Medical Education
      - Admissions
      - Assessments

    * Administrative offices
        - Finance
        - Personnel

Managing entities

Let's start by getting to work on an entity.

The Entities change list

In the Arkestra admin site, find *Contacts & people* > *Entities*. They are arranged in a tree reflecting the hierarchy of entities.

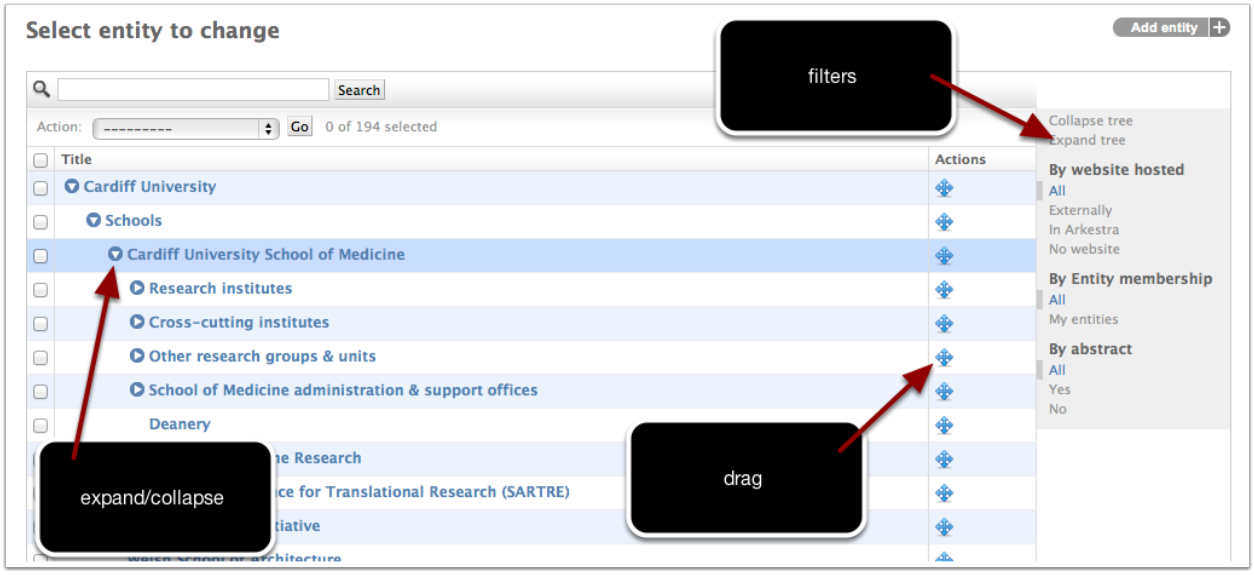

Fig. 2.1: The **Entities** change list

You can:

- **search** the list of Entities
- **filter** the list according to various criteria
- **expand and collapse** sections of the hiearchy using the controls in the window
- **move** entites by dragging them around within the tree

Either select an existing entity to edit, or create a new one if none exist.

Editing an entity

This is where we tell Arkestra what it needs to know about our entity.

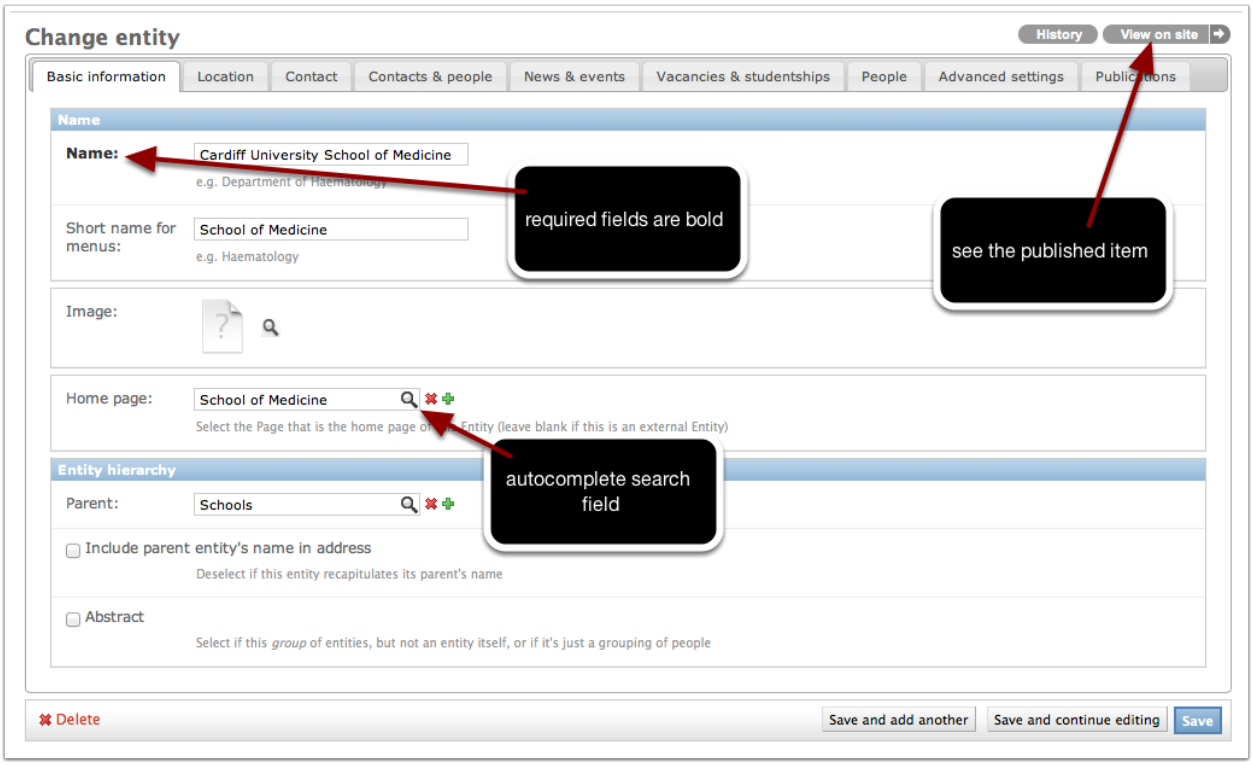

Fig. 2.2: The **Entity** change view - you'll find many of the same controls and features throughout Arkestra's administration interface.

Basic information

Name its official title, in full

Short name for menus a shorter version, if required

Image an image (for example a logo)

Home page The entity will likely have a number of pages associated with it in the **Pages** system that; select the root page of that section. Arkestra will know that *all* the pages in that section belong to this entity.

Parent The entity's parent in the hierarchy. You can also change this by dragging the entity around in the change list, but if the hierarchy contains hundreds of items, this is a more efficient way to do it.

Include parent entity's name in address Arkestra builds an entity's address automatically, and will include its parent's name in that automatically. Usually that makes sense (*IT Office, School of Medicine*, but sometimes it might look silly (*Cardiff University School of Medicine, Cardiff University*), so this allows us to turn it off.

Abstract Some entities are actual organisational entities, like the University or the Research Office. Others are just useful groupings, or *abstract entities*.

Location

Address This is there to show you how Arkestra will render the entity's address

Building where the entity is based

Building recapitulates entity name Sometimes, the building name and the entity name are so similar it would be silly to have them both in the address. If so, this field allows Arkestra to know that it shouldn't include them both in addresses

Precise location the place *inside* the building - floor, room number, etc

Precise location any note for potential visitors

Contact

Email address if the entity has its own email address

Phone contacts choose or add a suitable label; don't try to format numbers

Contacts & people, News & Events, Vacancies & Studentships

Arkestra will publish pages of various types automatically if required. The controls are very similar for them all, and also for other modules that might be added by other applications, and will look much like:

Publish a contacts and people page for this entity automatically if this entity should have its own contacts and people page

Title what the page should be called

Things you should never do

Arkestra has been deisgned to guide you towards the right way of doing things, and away from bad practices. As far as it can, it will warn or even reprimand you when necessary, but you still need to be aware of the right and wrong way to do things.

Warning signs

If you ever find yourself doing any of these things, it's an excellent sign that you are probably doing something wrong:

Entering information that is already in Arkestra

The whole point of Arkestra is to record, manage and publish information, automatically wherever possible. If you find yourself entering the same information twice, then you're not allowing Arkestra to help you by re-using it (it also means that you now have the information in two places, both of which will have to be kept up-to-date).

A good example is address information. You should almost never have to write out an address in Arkestra. Arkestra maintains information about places, including their addresses.

Inventing new ways of using it

Don't be creative with Arkestra. There's no guarantee that the clever new way of doing something you've discovered will continue to work in the future. It's much safer and more reliable to learn how Arkestra is *supposed* to work.

Finding 'solutions' to apparent problems or shortcomings

You're almost certain to discover shortcomings and even faults in Arkestra. Don't try to work around them. That will only ever produce a second-best, and uncertain, solution. It's much better to report them, so that Arkestra can be improved, and accommodate whatever need you have for it.

The worst things you can do

Some abuses are worse than others...

1. Write "here", or "click here", in link text.

CHAPTER 3

How it works

Arkestra templating

Placeholder & image sizing cues

Your main site template(s) should include some cues for the image sizing system.

Obviously, these should reflect the site's CSS.

The built-in *arkestra_utilities/templates/arkestra.html* sets some defaults, so if your own template extends that (certainly recommended while you are getting started) you will find that the values below are already set.

They are set using *{% with %}{% endwith %}* around the *{% block body_content %}*. If you want to override any of them in your own template, use *{% with %}{% endwith %}* just *inside* in your template's *{% block body_content %}*, the way *institute.html* does in the example project.

```
placeholder_width=960 # the main body, default for all placeholders
generic_main_width=523 # width of the main part of a news item, event, etc
sidebar_image_size="294x196" # images in news, events, etc sidebars
sidebar_map_size="296x100" # map thumbnail image in events etc sidebars
entity_map_size="445x100" # map thumbnail image for entites
person_map_size="445x100" # map thumbnail image for persons
entity_image_size="445x384" # main entity image
person_image_size="384x384" # person's main image
person_thumbnail_size="40x40" # person's thumbnail image in lists
place_image_size="627x418"  # a place's main image
place_map_size="294x182 # the small map on a place page
lightbox_max_dimension=600  # what's the biggest a lightbox can be?
plugin_thumbnail_size="75x75"
```

Other cues

Arkestra also provides template hooks for other variables, which can be set in the same *{% with %}*/*{% endwith %}* block.

```
slider_delay="4000" # time for each slide in the ImageSet slider
```

{% block main_page_body %}

Some views, such as for an entity's Contacts & People page or News & Events page, expect your templates to be lined up in the right way. For example:

- the view for a Contacts & People page will load *arkestra/entity_generic_lister_page.html*
- this extend the template that it gets from the entity's cms.Page: *{% extends entity.get_template %}*
- the site template in its *{% block main_page_body %}* will thus include the file that view has specified, containing the special stuff for that view: *{% include generic_lister_template %}*

So, anything in your template that should *not* end up in pages like Contacts & People should be *inside {% block main_page_body %}*, so that it is replaced.

Anything outside that block will appear on those pages too.

Using news & events

Events

The hierarchy of events

Events in Arkestra are maintained in a hierarchy, because that's what they are in real life. The stages of each year's Tour de France are the *children* of that year's race:

- **Tour de France 1969**
  - Stage 1
  - Stage 2
  - Stage 3
  - Stage 4
  - [... and so on]
- **Tour de France 1970**
  - Stage 1
  - Stage 2
  - Stage 3
  - Stage 4
  - [... and so on]

If we wanted, we could identify different events within each stage too - the Départ Fictif, the Départ Réel, the award ceremony - and Arkestra would happily deal with that too.

(Another typical example of a hierarchy would be the sessions in a conference.)

Actual events and series of events

This year's Tour de France is an *actual* event, as was last year's and as will be next's. It starts and ends.

But the Tour de France itself is a *series* of *actual* events. The *series* is the *parent* of each annual event.

- **Tour de France**
    - **Tour de France 1969**
        - Stage 1
        - Stage 2
        - Stage 3
        - Stage 4
        - [... and so on]
    - **Tour de France 1970**
        - [...]
    - **Tour de France 1971**
        - [...]
    - **Tour de France 1972**
        - [...]

We can't store a start or end date for the *series*, but there are still lots of other useful items of information that we might want to record about the series as a whole, or that apply to the *actual* events that are its children.

So Arkestra can record an event as being a *series*. Any event that doesn't have at least a start date is assumed to be a series (what else could it be?).

What gets shown in Events lists?

In a list of events, such as on a News & Events page, which items will be listed?

- actual events (not series events) that do not have an actual event parent

So, Tour de France 1969 will show up in the list, but not all its children. If someone wants to see details of all the children, they can visit the Tour de France 1969 page.

Similarly, we would want to show a conference, but not all of its sessions.

Each item in the list has this basic structure:

- *[title of series, link to series]* (if the event is part of series)
- *[title of event, link to event]* (if the event merits its own page)
- *[summary of event]*
- *[date]*
- *[venue]*

or for a concrete example:

- Public Health lecture series *[links to the page for the series]*
- "The health benefits & risks of milk" *[links to the page for this particular item]*
- A debate between two leading researchers
- 12th January, 19:00
- UHW Main Building

We don't always need to show all of these. *[title of series, link to series]* can't be shown if the item isn't part of a series, for example.

The image sizing system

The problem

Suppose we have a page with two columns, and an image, 500px wide, in the second column, which happens to be 500px wide itself.

At the moment, a 500px width is correct for that image, but any one of a number of things could change that, for example:

- if the editor decides the layout should no longer be two columns but three
- ... or that the text column should be twice the width of the image columns
- if the editor applies a border to the image
- the designer decides that the site template should be 200px wider
- the designer applies CSS to the site placing borders on all images

Any of these will mean that the 500px wide image will no longer fit neatly into its column.

One solution is for the editor to work out the new correct size for the image and set it accordingly. Of course, if this is in the process of working out a new page layout it could be a rather tedious; it could be more tedious still if it's the result of a site-wide change, when there could be hundreds or thousands of images to correct.

The other solution is for Arkestra to do it all automatically, which it does.

The solution

Arkestra can know about site templates, and use this information to work out the correct size of any image.

For example, suppose the body content area of in our example above is 960 pixels wide. Arkestra will know what the exact width of the image should be to fill it.

Using the default CSS and values:

- to occupy that second column, it will be be 48% of 960
- for a three-column layout, or if the text column should be twice the width: 30.6667%
- if it has a border class applied: reduce the width by 16 pixels.

These are just the simplest examples. If your image has been told to

- float right
- have a border

- and occupy 50% of a column
- which itself has a background tint
- and is two-fifths of the second column in a two-column layout
- and the page has a menu down the side that makes the content area narrower

Arkestra will take account of all that to work out the size of the image to the nearest pixel. Every time any of those things changes, when the page is rendered, the image will appear at exactly the right size.

Arkestra comes with sensible defaults for columns and borders, but they are easy to override if you want your own templates to have different values.

How it works

The image plugin (or video, or carousel, or whatever - referred to here as 'image', just to make life easier) is told what width to be.

If it's an absolute width or the native width of the image, then that's settled. But if it's one a width relative to the containing column, then Arkestra will need to make the calculation.

In summary, it looks like this: * the rendering plugin

- ***get_plugin_width* - if not None (native width) or negative (absolute width) then:**
  - ***get_placeholder_width***
    - *SimplePlaceholderWidthAdjuster* (the default *placeholder_width* adjuster)
  - ***calculate_container_width***
    - *get_plugin_ancestry*
    - **for each ancestor:**
      - run each registered *plugin_width* adjuster (some modify immediately, some modify later); defaults are:
      - *AutoSpaceFloat*
      - *ReduceForBackground*
      - *ColumnWidths*
      - *ImageBorders* - a *mark_and_modify* adjusters: it acts only once for the entire tree

Calculate the width of the placeholder

The *get_placeholder_width* function is called by the rendering plugin, such as *arkestra_image_plugin.cms_plugins.FilerImagePlugin*.

This obtains the default *placeholder_width* value from the context.

This is how:

- *placeholder_width* can be set by CMS_PLACEHOLDER_CONF["body"]["extra_content"]["width"]
- *placeholder_width* can be set by {% with placeholder_width=<some_value> %} around the placeholder
- failing to get *placeholder_width*, it looks for *width*, provided by [what? cms?]
- failing that, it just chooses 100 - useful for admin templates

get_placeholder_width then calls each registered *placeholder_width* adjuster. There is one included by default: *SimplePlaceholderWidthAdjuster*.

SimplePlaceholderWidthAdjuster

This examines the context for your clues on how placeholder widths should be adjusted.

Suppose that if the page has a menu down the side, then the placeholder width should be different - 749px, say. So it might look like:

{% with adjust_width=current_page.flags.local_menu width_adjuster="absolute" width_adjustment=749 %}

This means:

- adjust the width if current_page.flags.local_menu is found
- we want to use an absolute width
- which will be 749px

SimplePlaceholderWidthAdjuster inspects the context for an *adjust_width* variable. If found, it will also look for *adjuster* and *adjustment*.

Possible *adjuster* values:

- divider (divide the *placeholder_width* by *adjustment*)
- multiplier (multiply it)
- percent (the calculated value will be *adjustment* percent of *placeholder_width*)
- relative (add *adjustment* percent to *placeholder_width*)
- absolute (set the new value to *adjustment*)

Obviously it's up to you and your HTML/CSS how all these things work...

Anyway, now Arkestra knows how wide the placeholder is, and returns that.

Calculate image container width

Now we need to find out the width of the immediate container in which the image plugin finds itself.

It might be directly placed in the placeholder, or it might be within a text plugin within the placeholder, or in a deeper structure still.

So, the rendering function calls *calculate_container_width*.

First, this obtains all the ancestors in the plugin hierarchy tree, from *get_plugin_ancestry*.

Then, starting at the rootmost plugin, it will run each registered *plugin_width* adjuster.

There is one included by default: *KeyReducer*.

AutoSpaceFloat

AutoSpaceFloat uses a truth table to determine how to adjust the width of the container, depending upon whether not the width is set to automatic, whether the space-on-left or space-on-right classes have been used, and whether the image is floated.

Next it will examine the HTML of the plugin (using *BeautifulSoup*), and find where the next plugin is in the HTML structure. It will then examine the HTML structure of nested elements, from the root upwards.

For each element, it will run the *image_width* adjusters.

ReduceForBackground

The second allows for backgrounds - if elements with background tints also have padding, which they usually do, we need to allow for that.

The effect of this padding is cumulative - if three nested elements all have padding, then the reduction for the padding will need to be applied for each one.

ReduceForBackground by default tests for *tint* or *outline* in the element class, and applies a 16px width reduction.

This can be overridden in the template by using {% with %}:

- background_classes="some-class some-other-class" (space-separated values)
- background_reduction=16

ColumnWidths

The second of these calculates the column width.

ImageBorders

The final kind of adjuster is the *mark_and_modify* adjusters, which run two functions, one to mark the elements that need acting on, and one to act on them afterwards.

These inspect every element, but don't modify the width for every one - they only act once per plugin.

For example, even if several elements in the image plugin's ancestry have a border class on them, the image can only have one border.

ImageBorders by default tests for *image-borders* and *no-image-borders* in the element classes.

These tests add a key - *markers["has_borders"]* to the dictionary that looks after this.

Finally, after all the rest is done, *calculate_container_width* will run the *modify* functions of these adjusters.

The defaults can be overridden in the template by using {% with %}:

- image_border_class="some-class"
- no_image_border_class="some-other-class"
- image_border_reduction=16

At the end of all this, *calculate_container_width* returns the calculated width of the container of the image.

The links system

The links system is one of the most complex parts of Arkestra.

The ExternalLinks database

Saving an ExternalLink in Admin

ExternalLinkForm.clean() checks that the link is in order.

If the URL (which must be unique) already exists in the database, an error is raised; if the title already exists, a warning is raised (a duplicate title could be confusing but is not fatal).

When an object that can have an External URL is saved, the ModelAdmin.clean() of that object must call *links.admin.get_or_create_external_link()*, passing it various items of information:

- self.cleaned_data.get("input_url", None), # a manually entered url
- self.cleaned_data.get("external_url", None), # a url chosen with autocomplete
- self.cleaned_data.get("title"), # link title
- self.cleaned_data.get("summary"), # link description

links.admin.get_or_create_external_link() checks that information against the database:

- is the URL scheme of *input_url* or *external_url* permitted by *links.utils.check_urls()*?
- has an *input_url* been provided?
  - get_or_create an *ExternalLink* based on it

The links schema system

Any model can be registered with the links system, making it easy to search for - using the admin autocomplete search - and link to published instances of the model.

The registry system works in a similar way to django.contrib.admin.

A *link_schema.py* module is required - see the one in *contacts_and_people* for an example.

Any object we're going to link to needs a particular set of attributes. These can be built in to the model, or if they're not, we need to write a wrapper around the model to provide them.

A list of the needed attributes is in *links.schema_registry.ATTRIBUTES*.

The very simplest case

The model has all the needed attributes already; even the admin has the *search_fields* declared:

```
# import the model
from news_and_events.models import Event

# import the admin class
from news_and_events.admin import EventAdmin

# register it
schema.register(
    models.Event,
    search_fields=admin.EventAdmin.search_fields
    )
```

This works because *FormDefinition* does indeed happen to have a (required) *title* attribute, though we won't always be so lucky.

Now when we're using any class that inherits from *links.models.LinkMethodsMixin*, we'll be able to choose the type *FormDefinition*, and search through its instances for one to make a link to.

Nearly as simple

Perhaps the attributes are there, but we need to do a little more work to get hold of them:

```
schema.register(
    models.Event,
    search_fields=admin.EventAdmin.search_fields,
        # wrapper attributes
        title='name',
        description='some_foreign_key.title',
        heading='"Event"'
        )
```

Each named argument after search_fields represents a attribute the wrapper will have.

We can use a dotted path to traverse through object, starting from the linked object.

description='some_foreign_key.title' use the object's *some_foreign_key.title* for the description. Any attribute - a callable or property - will do.

heading="'Event"' use a static string for the heading under which objects of this type will be grouped in lists of links. Both *"'foo"'* and *"'bar"'* work.

Let's do some extra calculation

This works by providing a callable that takes one argument: the linked object It can be a lamda function or a external function:

```
def some_very_complicated_function(obj):
    r = []
    for x in obj.something.all():
            if x.y:
                    r.append(u"foo: %s" % x.bar)
            else:
                    r.append(u"stuff: %s" % x.bar)
    return "<br />".join(r)

schema.register(
    models.Event,
    search_fields=admin.EventAdmin.search_fields,
    title=lambda obj: u"%s (%s)" % (unicode(obj), obj.active_count),
    description=some_very_complicated_function
    )
```

Create our own *LinkWrapper* subclass

We can also create a *LinkWrapper* and provide it with the required attributes. Some of these might do some complex work for us, but here's a simple one first:

```
# import the model
from form_designer.models import FormDefinition

# declare a wrapper with a search field
class FormDefinitionLinkWrapper(LinkWrapper):
    search_fields = ['title',]

# register the wrapper
schema.register_wrapper(FormDefinition, FormDefinitionLinkWrapper)
```

The video system

The video plugin

What the plugin does when it renders

First, just as for Image plugins, we have to work out the width of the space available to us.

Given the width, we need to compare that with the available re-rendered versions of the video to see which is the best fit.

Now we find which versions of the video are available.

Arkestra generic models

Arkestra provides a flexible way to query your database for information and have it published - automatically, in the right time, in the right place.

This is a full description of how the news_and_events application does this.

models.py

Import the classes you'll need:

```
from arkestra_utilities.generic_models import ArkestraGenericPluginOptions,␣
↪ArkestraGenericModel
from arkestra_utilities.mixins import URLModelMixin
```

Inherit the ones you need into your new model class:

```
class NewsArticle(ArkestraGenericModel, URLModelMixin):
```

You don't need to inherit URLModelMixin, but it can be useful. URLModelMixin provides fields:

- slug
- external_url

and methods:

- __unicode__()
- get_absolute_url()

URLModelMixin is handy if your instances of your model will each have their own page on the site.

ArkestraGenericModel provides:

title the full title of the item

short_title a short version, for lists

summary a brief summary, used in lists and on the item's main page

body a PlaceholderField

image for the item's page, and to offer a little thumbnail image for lists

hosted_by the Entity that hosts or publishes the item

publish_to the other Entities to whose pages it should be published

please_contact a Person

importance so Arkestra can highlight it when appropriate

and @properties:

get_importance marks items as important, to help gather them together or highlight them in lists as required

is_uninformative the item bears little information of its own

get_template the template of the webpage of the entity of this item

- links
- external_url
- get_hosted_by
- get_website

And you can add whatever fields of your own that are required and ignore the ones that are not.

admin.py

Import some handy mixins:

```
from arkestra_utilities.admin_mixins import SupplyRequestMixin, AutocompleteMixin,␣
↪InputURLMixin, fieldsets
```

- SupplyRequestMixin supplies the context to the admin - you might have a need for it
- AutocompleteMixin
- InputURLMixin
- fieldsets: some handy predefined fieldsets

Define the admin form and class, and do the usual things with them:

```
class NewsArticleForm(InputURLMixin):
    class Meta:
        model = NewsArticle


class NewsArticledmin(SupplyRequestMixin, AutocompleteMixin,␣
↪ModelAdminWithTabsAndCMSPlaceholder):
    related_search_fields = ['hosted_by', 'external_url',] # autocomplete on these␣
↪fields
```

```
    def _media(self):
        return super(AutocompleteMixin, self).media +␣
↪super(ModelAdminWithTabsAndCMSPlaceholder, self).media
    media = property(_media)
```

urls.py

To `urls.py` add a url pattern:

```
(r"^news/(?P<slug>[-\w]+)/$", "news_and_events.views.newsarticle"),
```

views.py

We need to provide the view the urlpattern points to:

```
def newsarticle(request, slug):
    """
    Responsible for publishing news article
    """
    newsarticle = get_object_or_404(NewsArticle, slug=slug)

    return render_to_response(
        "news_and_events/newsarticle.html",
        {
        "newsarticle":newsarticle,
        "entity": newsarticle.hosted_by,
        "meta": {"description": newsarticle.summary,}
        },
        RequestContext(request),
        )
```

news_and_events/newsarticle.html

The best thing to do is to have a look at the actual `news_and_events/newsarticle.html`.

Same salient points:

- {% extends newsarticle.get_template %} - see ArkestraGenericModel.get_template, above
- page furniture, such as the metadata, will be handled by the template it extends

managers.py

It's useful to give your model a manager. You don't *need* to, but it helps keep things tidy, and we'll use one in this example. In particular, if you want to make use of the ArkestraGenericPlugin, that makes use of a `get_items()` method on your manager.

Inherit the generic model manager:

```
from arkestra_utilities.managers import ArkestraGenericModelManager
```

At present this only contains:

```
def get_by_natural_key(self, slug):
    return self.get(slug=slug)
```

but in the future it might acquire more.

Define your manager and give it a `get_items()` method:

```
class NewsArticleManager(ArkestraGenericModelManager):
    def get_items(self, instance):
                # just for now, we will return all the objects of this model
        return self.model.objects.all()
```

`get_items()` can be very complex - see the news_and_events.EventManager for a particularly complex example.

The `instance` argument for the manager is actually an instance of the plugin model class, which functions as a reasonably convenient API.

Go back to your model and add an attribute so it knows about the manager:

```
objects = NewsArticleManager()
```

cms_plugins.py

The simplest kind of plugin isn't even configurable. You just insert it into your content, and let it do its work. We'll start with one of those:

```
from arkestra_utilities.generic_models import ArkestraGenericPlugin
```

ArkestraGenericPlugin refers throughout to `instance`.

`instance` is the class that defines the behaviour of the plugin in this particular instance. If the plugin is configurable, the instance is the model instance as set up in the Admin; if not, it's just an instance of the same model class created for the purpose, but not stored in the database. If we haven't even created sucha model class ourselves, it will be an instance of `cms.models.pluginmodel.CMSPlugin`.

ArkestraGenericPlugin provides a number of methods, mostly called by `render()`:

`render(self, context, instance, placeholder)`

- works out the entity
- assumes the `type` of the instance is `plugin` if not stated otherwise (e.g. a menu generator, a main page generator)
- changes the render_template from `arkestra/universal_plugin_lister.html` if required
- calls `set_defaults()` to set some sensible defaults which may or may not be overridden
- calls `get_items()` to get items in a list of lists, called `lists`.
- calls `add_link_to_main_page()` to see if we need a link to a main page (e.g. the main news and events page)
- calls `add_links_to_other_items()` to see if we should provide links to archives etc
- calls `set_limits_and_indexes()` to work out whether we need indexes, or how to truncate lists of items
- calls `determine_layout_settings()` to set rows/columns and classes for items in the lists
- calls `set_layout_classes()` to work out the overall structure (rows/columns) of the plugin output

Everything it needs to set for the overall information about what's going on in the plugin is set as an attribute of `instance`, which is then passed to the template as `everything`. `lists` is made an attribute of `instance`.

`get_items()` isn't provided by ArkestraGenericPlugin, except as a dummy that sets an empty `lists` - it needs to be provided by whatever subclasses it:

```
self.lists = []
```

This is because ArkestraGenericPlugin won't have any idea how to get items - it doesn't know about content.

```
class CMSNewsAndEventsPlugin(ArkestraGenericPlugin, CMSPluginBase):
    # set text_enabled, admin_preview, render_template if the ArkestraGenericPlugin␣
↪default are not suitable

    # provide an icon for the admin interface
            def icon_src(self, instance):
        return "/static/plugin_icons/news_and_events.png"

plugin_pool.register_plugin(CMSNewsAndEventsPlugin)
```

You should now be able to insert the plugin into a placeholder, and examine its output - but there won't be anything in there yet, because `get_items()` returns `[]`.

So let's add a method to our plugin:

```
def get_items(self, instance):
    # call the base get_items() to set up our self.lists
            super(CMSPublicationsPlugin, self).get_items(instance)

    # create a dict to store information about the news articles
            news_articles = {}

    # put the actual items in it
            news_articles["items"] = NewsArticle.objects.all()

    # will the plugin publish links to other items (more news, news archive)?
            news_articles["links_to_other_items"] = self.news_style_other_links

    # will the plugin publish links to other items (more news, news archive)?
            news_articles["heading_text"] = instance.news_heading_text

            # what template will each item in this list use?
            news_articles["item_template"] = "arkestra/universal_plugin_list_item.html
↪"


    self.lists.append(news_articles)
```

And if we haven't changed `render_template`, it will use `arkestra/universal_plugin_lister.html`. Use `arkestra/universal_plugin_lister.html` as a guide to writing your template.

add_link_to_main_page()

To be completed

urls.py

We can use the architecture we have been working with to create an automatic page, for each entity, for this - or these - generic models.

We need a URL pattern for this:

```
# named entities' news
(r"^news/(?P<slug>[-\w]+)/$", "news.views.news"),

# base entity's news
(r'^news/$', "news.views.news"),
```

views.py

models.py (again)

Now let's create a plugin that we can use to list a number of the items in the model we have created.

We have already imported arkestra_utilities.ArkestraGenericPluginOptions. This provides:

```
* entity: the entity whose items we'll publish (can usually be left blank; Arkestra␣
↪will work out what to do)
* layout: if there are multiple lists (e.g. news and events), will they be stacked or␣
↪side-by-side?
* format: title only? details? image?
* heading_level: above the list there'll be a heading
* order_by: date alone, or rank by importance too?
* list format: horizontal or vertical
* group dates: group lists into sublists (of months, usually)
* limit_to: how many items - leave blank for no limit
```

```
class NewsAndEventsPlugin(CMSPlugin, ArkestraGenericPluginOptions):
```

Note that this plugin can handle both news and events.

And let's add:

```
display = models.CharField("Show", max_length=25,choices = DISPLAY, default = "news␣
↪events")
show_previous_events = models.BooleanField()
news_heading_text = models.CharField(max_length=25, default="News")
events_heading_text = models.CharField(max_length=25, default="Events")
```

class NewsAndEventsPlugin(CMSPlugin, ArkestraGenericPluginOptions):

This is in effect the model for the plugin, and will inherit:

form and admin

from arkestra_utilities.generic_models import ArkestraGenericPlugin, ArkestraGenericPluginForm from arkestra_utilities.mixins import AutocompleteMixin

class PublicationsPluginForm(ArkestraGenericPluginForm, forms.ModelForm): pass

class CMSPublicationsPlugin(UniversalPlugin, AutocompleteMixin, CMSPluginBase): model = PublicationsPlugin name = _("Publications") form = PublicationsPluginForm auto_page_attribute = "auto_publications_page" auto_page_slug = "publications" auto_page_menu_title = "publications_page_menu_title" # fieldsets = (# (None, { # 'fields': (('display', 'layout', 'list_format',), ('format', 'order_by', 'group_dates',), 'limit_to') # }), # ('Advanced options', { # 'classes': ('collapse',), # 'fields': ('entity', 'heading_level', ('news_heading_text', 'events_heading_text'), ('show_previous_events',),) # }), #)

autocomplete fields related_search_fields = ['entity',]

def get_items(self, instance): self.lists = []

this_list = {"model": Publication,} this_list["items"] = Pub.objects.get_items(instance) this_list["links_to_other_items"] = self.news_style_other_links this_list["heading_text"] = instance.news_heading_text this_list["item_template"] = "arkestra/universal_plugin_list_item.html" # the following should *also* check this_list["links_to_other_items"] - # but then get_items() will need to call self.add_links_to_other_items() itself # this will then mean that news and events pages show two columns if one has links to other items if this_list["items"]:

self.lists.append(this_list)

def icon_src(self, instance): return "/static/plugin_icons/publications_plugin.png"

Menu

Every Entity in the system that has Recordings should have a menu item where they're listed.

For now we will just hardcode a little routine into our menu, contacts_and_people.menu, at the comment "# insert nodes for this Entity":

self.create_new_node(title = "Recordings", url = node.entity.get_auto_page_url("recordings"), # i.e. /url_of_entity/recordings parent = node,)

We'll make this more sophisticated later.

URL

We need a URL pattern to match that, so you'll need:

named entities' recordings (r"^recordings/(?P<slug>[-w]+)/$", "recordings.views.recordings"),

base entity's vacancies and studentships (r'^recordings/$', "recordings.views.recordings"),

Views

Your URL is looking for a view:

class MyPluginPublisher(ArkestraGenericPlugin, AutocompleteMixin, CMSPluginBase):

This is in effect the admin for the plugin. Its render() method is what publishes the output. It will inherit:

How to use and abuse this plugin:

first create an instance of the plugin model:

instance = NewsAndEventsPlugin()

set the attributes as required:

instance.display = "events" instance.type = "for_place" instance.place = self instance.view = "current" instance.format = "details image"

render it to get back the items you want in instance.lists, if you have the context:

CMSNewsAndEventsPlugin().render(context, instance, None)

alternatively (this is used in the menus, for example):

plugin = CMSNewsAndEventsPlugin() plugin.get_items(instance) plugin.add_links_to_other_items(instance) ... and any operations tests as required

and the NewsAndEventsPlugin() needs to have the lists attribute of CMSNewsAndEventsPlugin()

Migrating from an older installation

If you have been using a version of Arkestra prior to the publishing of version 2.0 on Github, you have a bit of work to do.

This may look unpleasant, but it's been tested and is mainly a matter of making sure that you paste into the right place.

Before you do anything else

Work on a copy of your repository and database. You were going to do that anyway, right?

Clear out the old migration records

Find your south_migrationhistory table and clear out the old migration records. We're going to re-create these, so you need to:

- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’arkestra_image_plugin’;
- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’contacts_and_people’;
- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’links’;
- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’news_and_events’;
- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’vacancies_and_studentships’;
- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’video’;

Trash your old migrations

Now, for each Arkestra application, trash its migrations directory (because we want to get rid of the old messy trail of migrations, and have just a single clean initial migration for each).

Do this in the Arkestra directory:

- *rm -r arkestra_image_plugin/migrations contacts_and_people/migrations links/migrations news_and_events/migrations vacancies_and_studentships/migrations video/migrations*

Create new migrations

Then create new, clean initial migrations for each one.

Make sure you're in the project directory:

- *python manage.py convert_to_south arkestra_image_plugin –delete-ghost-migrations*

- *python manage.py convert_to_south contacts_and_people*
- *python manage.py convert_to_south links*
- *python manage.py convert_to_south news_and_events*
- *python manage.py convert_to_south vacancies_and_studentships*
- *python manage.py convert_to_south video*

Now your migrations should match your models, and your database records should indicate that they match.

Move your migrations

In a moment, you'll pull in the new version of Arkestra, but you don't want to overwrite the new migrations you just created, so put them in a safe place:

Do this in the Arkestra directory:

- mv *arkestra_image_plugin/migrations arkestra_image_plugin/my_migrations*
- mv *contacts_and_people/migrations contacts_and_people/my_migrations*
- mv *links/migrations links/my_migrations*
- mv *news_and_events/migrations news_and_events/my_migrations*
- mv *vacancies_and_studentships/migrations vacancies_and_studentships/my_migrations*
- mv *video/migrations video/my_migrations*

Checkout the new version of Arkestra

Switch to the branch of Arkestra with cleaned up migrations.

Fetch updated references from Github:

- *git fetch origin*

Checkout the reference clean-migrations branch.

- git *checkout clean-migrations*

Replace its migrations with the ones you created

- *rm -r arkestra_image_plugin/migrations contacts_and_people/migrations links/migrations news_and_events/migrations vacancies_and_studentships/migrations video/migrations*
- *mv arkestra_image_plugin/my_migrations arkestra_image_plugin/migrations*
- *mv contacts_and_people/my_migrations contacts_and_people/migrations*
- *mv links/my_migrations links/migrations*
- *mv news_and_events/my_migrations news_and_events/migrations*
- *mv vacancies_and_studentships/my_migrations vacancies_and_studentships/migrations*
- *mv video/my_migrations video/migrations*

Now you should have:

- the new Arkestra models

- a set of migrations matching your database tables

Create migrations to get from your tables to the new models

Make sure you're in the project directory:

- *python manage.py schemamigration –auto arkestra_image_plugin*
- *python manage.py schemamigration –auto contacts_and_people*
- *python manage.py schemamigration –auto links*
- *python manage.py schemamigration –auto news_and_events*
- *python manage.py schemamigration –auto vacancies_and_studentships*
- *python manage.py schemamigration –auto video*

For any models where your previous version differed from the new, you'll now have a second migration to get from old to new.

Apply the new migrations

It's always sensible to use –db-dry-run first to check:

- *python manage.py migrate –db-dry-run*

then if that seems ok:

- *python manage.py migrate*

Now your database tables and models are up-to-date!

Get back to the Arkestra codebase

Do this in the Arkestra directory - be warned, it will delete everything it finds there that wasn't in the branch you checked out :

- *git clean -dxf*

Clear out the migration records (again)

Once again, find your south_migrationhistory table and clear out the relevant migration records. We're going to re-create these, so you need to:

- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’arkestra_image_plugin’;
- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’contacts_and_people’;
- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’links’;
- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’news_and_events’;
- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’vacancies_and_studentships’;
- DELETE FROM *south_migrationhistory* WHERE ‘app_name‘=’video’;

Fake the migrations

Back to the project directory:

- *python manage.py migrate –fake arkestra_image_plugin*
- *python manage.py migrate –fake contacts_and_people*
- *python manage.py migrate –fake links*
- *python manage.py migrate –fake news_and_events*
- *python manage.py migrate –fake vacancies_and_studentships*
- *python manage.py migrate –fake video*

Finally, all the following should be in agreement with each other:

- models
- database tables
- migrations
- south's database records of applied migrations

Apply any newer migrations

At the moment, your code and database are up-to-date with the 2.0 release. But, things might have moved on since then. There could be new migrations in master, or another branch.

So, in the Arkestra directory:

- *git checkout master* [or the branch you want]

Back to the project directory:

- *python manage.py migrate*

And hopefully, that will be that!

Indices and tables

- genindex
- modindex
- search